Code Clones

SW Maintenance and Evolution

Emanuel Giger

University of Zurich
Department of Informatics

software evolution & architecture lab


Code Clones are similar segments of source code found in different places of a system

Two Fundamental Questions?

How do we define similar? When do we consider code segments to be similar?

How do we detect similar source code segments?

Two Fundamental Questions?

Both problems are surprisingly difficult

Find *all* clones but avoid *false positives*

***Fast* and *efficient* : Software systems are potentially large**

1. Exact Copies

```
if(m_swapChain){
    m_swapChain->SetFullscreenState(false, NULL);
}

if(m_rasterState){
    m_rasterState->Release();
    m_rasterState = 0;
}

if(m_depthStencilView){
    m_depthStencilView->Release();
    m_depthStencilView = 0;
}

if(m_depthStencilState){
    m_depthStencilState->Release();
    m_depthStencilState = 0;
}

if(m_depthStencilBuffer){
    m_depthStencilBuffer->Release();
    m_depthStencilBuffer = 0;
}

if(m_renderTargetView){
    m_renderTargetView->Release();
    m_renderTargetView = 0;
}
```


```
if(m_swapChain){
    m_swapChain->SetFullscreenState(false, NULL);
}

if(m_rasterState){
    m_rasterState->Release();
    m_rasterState = 0;
}

if(m_depthStencilView){
    m_depthStencilView->Release();
    m_depthStencilView = 0;
}

if(m_depthStencilState){
    m_depthStencilState->Release();
    m_depthStencilState = 0;
}

if(m_depthStencilBuffer){
    m_depthStencilBuffer->Release();
    m_depthStencilBuffer = 0;
}

if(m_renderTargetView){
    m_renderTargetView->Release();
    m_renderTargetView = 0;
}
```

2. Parameterized Copies

```
// Set up the description of the static vertex buffer.
vertexBufferDesc.Usage = D3D11_USAGE_DEFAULT;
vertexBufferDesc.ByteWidth = sizeof(VertexType) * m_vertexCount;
vertexBufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;

// Give the subresource structure a pointer to the vertex data.
vertexData.pSysMem = vertices;
vertexData.SysMemPitch = 0;
vertexData.SysMemSlicePitch = 0;

// Now create the vertex buffer.
result = device->CreateBuffer(&vertexBufferDesc, &vertexData, &m_vertexBuffer);
```

```
// Set up the description of the static index buffer.
indexBufferDesc.Usage = D3D11_USAGE_DEFAULT;
indexBufferDesc.ByteWidth = sizeof(unsigned long) * m_indexCount;
indexBufferDesc.BindFlags = D3D11_BIND_INDEX_BUFFER;

// Give the subresource structure a pointer to the index data.
indexData.pSysMem = indices;
indexData.SysMemPitch = 0;
indexData.SysMemSlicePitch = 0;

// Create the index buffer.
result = device->CreateBuffer(&indexBufferDesc, &indexData, &m_indexBuffer);
```

3. Extended Copies

```
while (unsorted){
      unsorted = false;
      for (int i=0; i < x.length-1; i++)
        if (x[i] > x[i+1]) {
          temp       = x[i];
          x[i]       = x[i+1];
          x[i+1]     = temp;
          unsorted = true;
        }
    }
```

```
while (unsorted){
      unsorted = false;
      for (int i=0; i < x.length-1; i++)
        if (x[i] > x[i+1]) {
            try{
              temp      = x[i];
              x[i]      = x[i+1];
              x[i+1]    = temp;
            }
            catch(IndexException){
              ErrorLogger.("...");
              return false;
            }
          unsorted = true;
        }
    }
```

Code Clones Types

Cloned code segments segments can be found in *different files*, in the *same files* but in *different methods*, or in the *same method*

Segments must contain some *kind of logic or structure* that can be abstracted

```
....
computeVectorCrossProduct(v1, v2);
....
```

```
....
setIP("182.89.34.21");
....
```

```
....
computeVectorCrossProduct(v2, v3);
....
```

```
....
setIP("182.89.34.21");
....
```

Code Clones Types

Cloned code segments segments can be found in *different files*, in the *same files* but in *different methods*, or in the *same method*

Segments must contain some *kind of logic or structure* that can be abstracted

```
....
computeVectorCrossProduct(v1, v2);
....
```

Most likely not a clone

```
....
computeVectorCrossProduct(v2, v3);
....
```

```
....
setIP("182.89.34.21");
....
```

```
....
setIP("182.89.34.21");
....
```

Code Clones Types

Cloned code segments segments can be found in *different files*, in the *same files* but in *different methods*, or in the *same method*

Segments must contain some *kind of logic or structure* that can be abstracted

```
....
computeVectorCrossProduct(v1, v2);
....
```

Most likely not a clone

```
....
computeVectorCrossProduct(v2, v3);
....
```

```
....
setIP("182.89.34.21");
....
```

Potential clone

```
....
setIP("182.89.34.21");
....
```


Copied artifacts range from expressions, to functions, to data structures, and to entire subsystems.

Why Code Clones?

- Ctrl&C and Ctrl&V: Simple and fast way of code reuse, “templating”
- No time to factor out useful code
- Unexperienced developers
- Technological constraints: Frameworks, programming language (e.g., no polymorphism)
- Architectural constraints
- ...

Bad Clones Bad Clones

- Code bloat
- Doubles, triples, quadruples, ... the effort maintenance: *Code reading* and *modification*
- Copied defects
- Risk of *inconsistent changes* or “*forgotten*” clone segments
- Increases testing efforts when clones are scattered throughout different files/methods

Simply, it is a question regarding the aesthetic of code

How much Code Clones are out there?

- 8 to 12% in normal industrial code
- gcc: 8.7%, LOC 460k
- Database Server: 36.4%, LOC 245k
- X Windows: 19%
- Payroll Software: 59.3%
- Mostly small clones < 25 LOC

Active Research Topic

Code Clone Detection

Consequences of clones to maintenance

Tool support

Visualization of Code Clones

Characteristics of Code Clones

Clones considered harmful? - Studies are *inconclusive*

"Cloning Considered Harmful" Considered Harmful

Cory Kapser and Michael W. Godfrey
Software Architecture Group (SWAG)
David R. Cheriton School of Computer Science, University of Waterloo
{cjkapser, migod}@uwaterloo.ca

Abstract

Current literature on the topic of duplicated (cloned) code in software systems often considers duplication harmful to the system quality and the reasons commonly cited for duplicating code often have a negative connotation. While these positions are sometimes correct, during our case studies we have found that this is not universally true, and we have found several situations where code duplication seems to be a reasonable or even beneficial design option. For example, a method of introducing experimental changes to core subsystems is to duplicate the subsystem and introduce changes there in a kind of sandbox testbed. As features mature and become stable within the experimental subsystem, they can then be introduced gradually into the stable code base. In this way risk of introducing instabilities in the stable version is minimized. This paper describes several patterns of cloning that we have encountered in our case studies and discusses the advantages and disadvantages associated with using them.

1. Introduction

It is believed that most large software systems contain a non-trivial amount of redundant code. Often referred to as code clones, these segments of code typically involve 10–15% of the source code [24, 25]. Code clones can arise through a number of different activities. For example, intentional clones may be introduced through direct *"copy-and-pasting"* of code. Unintentional clones on the other hand may be the manifestation of programming idioms related to the language or libraries the developers are using.

In much of the literature on the topic [2, 7, 12, 21, 22, 27, 28], cloning is considered harmful to the quality of the source code. Code clones can cause additional maintenance effort. Changes to one segment of code may need to be propagated to several others, incurring unnecessary maintenance costs [15]. Locating and maintaining these clones pose additional problems if they do not evolve synchronously. With this in mind, methods for automatic refactoring have been suggested [4, 7], and tools specifically to aid developers in the manual refactoring of clones have also been developed [19].

There is no doubt that code cloning is often an indication of sloppy design and in such cases should be considered to be a kind of development "bad smell". However, we have found that there are many instances where this is simply not the case. For example, cloning may be used to introduce experimental optimizations to core subsystems without negatively effecting the stability of the main code. Thus, a variety of concerns such as stability, code ownership, and design clarity need to be considered before any refactoring is attempted; a manager should try to understand the reason behind the duplication before deciding what action (if any) to take. [1]

This paper introduces eight cloning patterns that we have uncovered during case studies on large software systems, some of which we reported in [23, 24, 25]. These patterns present both good and bad motivations for cloning, and we discuss both the advantages and disadvantages of these patterns of cloning in terms of development and maintenance. In some cases, we identify patterns of cloning that we believe are beneficial to the quality of the system. From our observations we have found that refactoring may not be the best solution in all patterns of cloning. Tools need to be developed to aid the synchronous maintenance of clones within a software system, such as Linked Editing presented by Toomim et al. [29].

This paper introduces the notion of categorizing high level patterns of cloning in a similar fashion to the cataloging of design patterns [14] or anti-patterns [8]. There are several benefits that can be gained from this characterization of cloning. First, it provides a flexible framework on top of which we can document our knowledge about how and why cloning occurs in

[1] A simple (but trivial) example is the title of this paper. Although there is a kind of duplication in the wording, no "refactoring" of the title would carry the same connotations as the original statement.

Do Code Clones Matter?

Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, Stefan Wagner
Institut für Informatik, Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
{juergens,deissenb,hummelb,wagnerst}@in.tum.de

Abstract

Code cloning is not only assumed to inflate maintenance costs but also considered defect-prone as inconsistent changes to code duplicates can lead to unexpected behavior. Consequently, the identification of duplicated code, clone detection, has been a very active area of research in recent years. Up to now, however, no substantial investigation of the consequences of code cloning on program correctness has been carried out. To remedy this shortcoming, this paper presents the results of a large-scale case study that was undertaken to find out if inconsistent changes to cloned code can indicate faults. For the analyzed commercial and open source systems we not only found that inconsistent changes to clones are very frequent but also identified a significant number of faults induced by such changes. The clone detection tool used in the case study implements a novel algorithm for the detection of inconsistent clones. It is available as open source to enable other researchers to use it as basis for further investigations.

1. Clones & correctness

Research in software maintenance has shown that many programs contain a significant amount of duplicated (cloned) code. Such cloned code is considered harmful for two reasons: (1) multiple, possibly unnecessary, duplicates of code increase maintenance costs and, (2) inconsistent changes to cloned code can create faults and, hence, lead to incorrect program behavior [20, 29]. While clone detection has been a very active area of research in recent years, up to now, there is no thorough understanding of the degree of harmfulness of code cloning. In fact, some researchers even started to doubt the harmfulness of cloning at all [17].

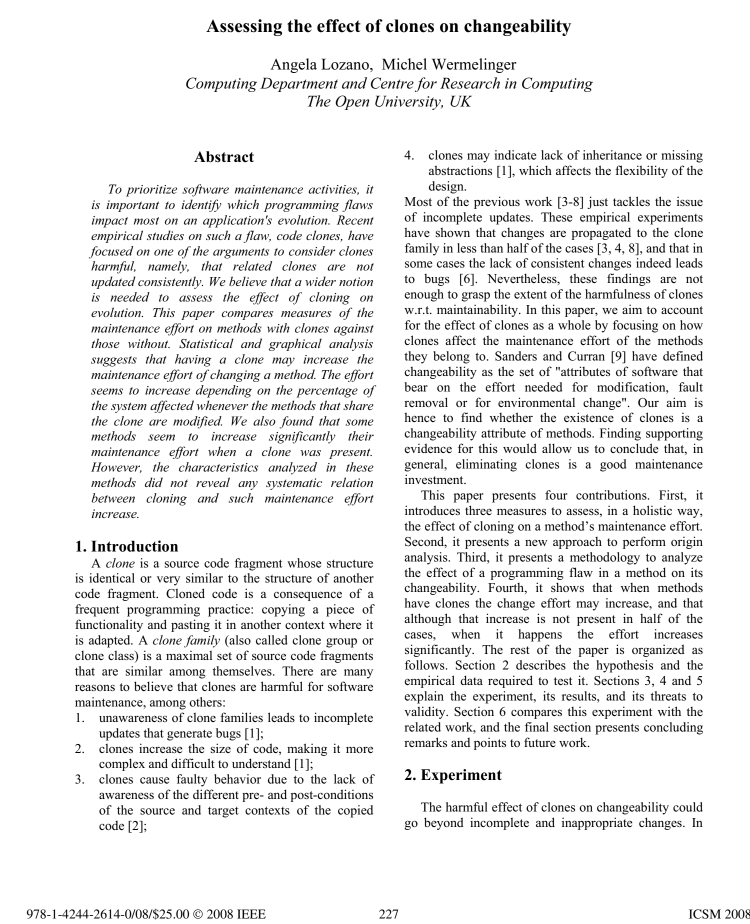
To shed light on the situation, we investigated the effects of code cloning on program correctness. It is important to understand, that clones do not directly cause faults but inconsistent changes to clones can lead to unexpected program behavior. A particularly dangerous type of change to cloned code is the *inconsistent bug fix*. If a fault was found in cloned code but not fixed in *all* clone instances, the system is likely to still exhibit the incorrect behavior. To illustrate this, Fig. 1 shows an example, where a missing null-check was retrofitted in only one clone instance.

This paper presents the results of a large-scale case study that was undertaken to find out (1) if clones are changed inconsistently, (2) if these inconsistencies are introduced intentionally and, (3) if unintentional inconsistencies can represent faults. In this case study we analyzed three commercial systems written in C#, one written in Cobol and one open-source system written in Java. To conduct the study we developed a novel detection algorithm that enables us to detect inconsistent clones. We *manually* inspected about 900 clone groups to handle the inevitable false positives and discussed each of the over 700 inconsistent clone groups with the *developers* of the respective systems to determine if the inconsistencies are intentional and if they represent faults. Altogether, around 1800 individual clone group assessments were manually performed in the course of the case study. The study lead to the identification of 107 faults that have been confirmed by the systems' developers.

Research Problem Although most previous work agrees that code cloning poses a problem for software maintenance, "there is little information available concerning the impacts of code clones on software quality" [29]. As the consequences of code cloning on program correctness, in particular, are not fully understood today, it remains unclear how *harmful* code clones really are. We consider the absence of a thorough understanding of code cloning precarious for software engineering research, education and practice.

Contribution The contribution of this paper is twofold. First, we extend the existing empirical knowledge by a case study that demonstrates that clones get changed inconsistently and that such changes can represent faults. Second, we present a novel suffix-tree based algorithm for the detection of inconsistent clones. In contrast to other algorithms for the detection of inconsistent clones, our tool suite is made available for other researchers as open source.

ICSE'09, May 16–24, 2009, Vancouver, Canada
978-1-4244-3452-7/09/$25.00 © 2009 IEEE



Assessing the effect of clones on changeability

Angela Lozano, Michel Wermelinger
Computing Department and Centre for Research in Computing
The Open University, UK

Abstract

To prioritize software maintenance activities, it is important to identify which programming flaws impact most on an application's evolution. Recent empirical studies on such a flaw, code clones, have focused on one of the arguments to consider clones harmful, namely, that related clones are not updated consistently. We believe that a wider notion is needed to assess the effect of cloning on evolution. This paper compares measures of the maintenance effort on methods with clones against those without. Statistical and graphical analysis suggests that having a clone may increase the maintenance effort of changing a method. The effort seems to increase depending on the percentage of the system affected whenever the methods that share the clone are modified. We also found that some methods seem to increase significantly their maintenance effort when a clone was present. However, the characteristics analyzed in these methods did not reveal any systematic relation between cloning and such maintenance effort increase.

1. Introduction

A *clone* is a source code fragment whose structure is identical or very similar to the structure of another code fragment. Cloned code is a consequence of a frequent programming practice: copying a piece of functionality and pasting it in another context where it is adapted. A *clone family* (also called clone group or clone class) is a maximal set of source code fragments that are similar among themselves. There are many reasons to believe that clones are harmful for software maintenance, among others:

1. unawareness of clone families leads to incomplete updates that generate bugs [1];
2. clones increase the size of code, making it more complex and difficult to understand [1];
3. clones cause faulty behavior due to the lack of awareness of the different pre- and post-conditions of the source and target contexts of the copied code [2];
4. clones may indicate lack of inheritance or missing abstractions [1], which affects the flexibility of the design.

Most of the previous work [3-8] just tackles the issue of incomplete updates. These empirical experiments have shown that changes are propagated to the clone family in less than half of the cases [3, 4, 8], and that in some cases the lack of consistent changes indeed leads to bugs [6]. Nevertheless, these findings are not enough to grasp the extent of the harmfulness of clones w.r.t. maintainability. In this paper, we aim to account for the effect of clones as a whole by focusing on how clones affect the maintenance effort of the methods they belong to. Sanders and Curran [9] have defined changeability as the set of "attributes of software that bear on the effort needed for modification, fault removal or for environmental change". Our aim is hence to find whether the existence of clones is a changeability attribute of methods. Finding supporting evidence for this would allow us to conclude that, in general, eliminating clones is a good maintenance investment.

This paper presents four contributions. First, it introduces three measures to assess, in a holistic way, the effect of cloning on a method's maintenance effort. Second, it presents a new approach to perform origin analysis. Third, it presents a methodology to analyze the effect of a programming flaw in a method on its changeability. Fourth, it shows that when methods have clones the change effort may increase, and that although that increase is not present in half of the cases, when it happens the effort increases significantly. The rest of the paper is organized as follows. Section 2 describes the hypothesis and the empirical data required to test it. Sections 3, 4 and 5 explain the experiment, its results, and its threats to validity. Section 6 compares this experiment with the related work, and the final section presents concluding remarks and points to future work.

2. Experiment

The harmful effect of clones on changeability could go beyond incomplete and inappropriate changes. In

978-1-4244-2614-0/08/$25.00 © 2008 IEEE

ICSM 2008

Not universally bad, some times clones makes sense, e.g, sandbox testbed

Empirical evidence of inconsistent changes that lead to faults

Some Indications, but no systematic relation between code clones and maintenance effort

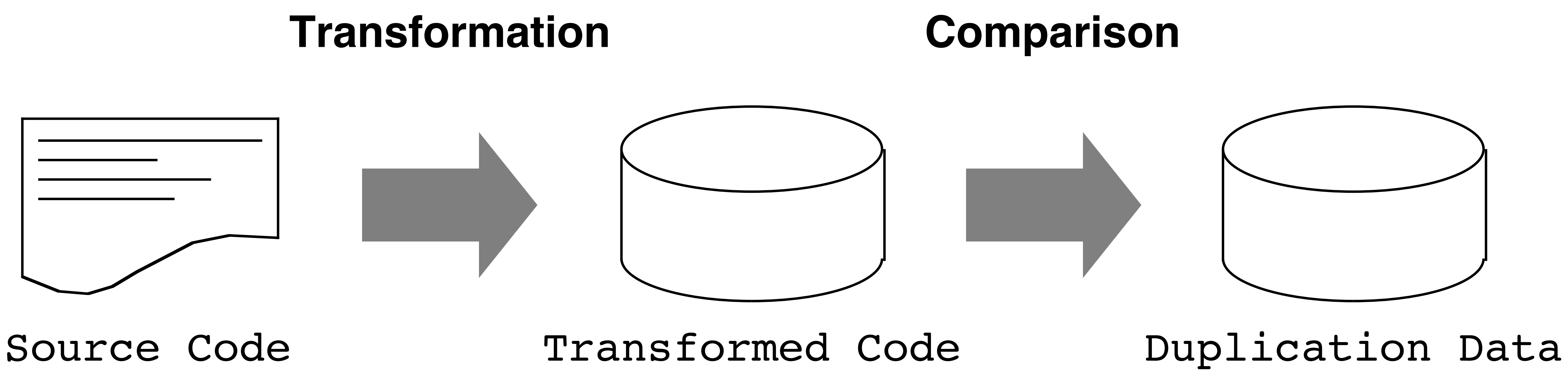
Code Clone Detection

Non-trivial Problem: *No a priori* knowledge about which code was cloned, its amount, its granularity

How to find all cloned segments among all possible pairs of segments: *Avoid computational complexity*

Define an appropriate *similarity measure*: Some clones are similar, but not identical - abstract from those differences

Code Clone Detection



Code Clone Detection

Author	**Level**	**Transformed Code**	**Comparison Technique**
Johnsen, 1994	Lexical	Substrings	String-Matching
Ducasse, 1999	Lexical	Normalized Strings	String-Matching
Baker, 1997	Syntactical	Token Strings	String-Matching
Mayrand, 1996	Syntactical	Metric Tuples	Discrete comparison
Kontogiannis, 1997	Syntactical	Metric Tuples	Euclidean distance
Baxter, 1998	Syntactical	AST-Representation	Tree-Matching

Token Based Clone Detection

Code is *tokenized* by set of of lexical rules (Lexical Analysis)

Token Based Clone Detection

Code is *tokenized* by set of of lexical rules (Lexical Analysis)

→

Token sequence is *transformed/normalized* by a set of rules

Rules Exmaples:

C++ Rules:

Remove namespace: std::vector **to** vector

Remove template parameters: vector<int> **to** vector

Java Rules:

Remove modifiers: protected void foo() **to** foo()

Token Based Clone Detection

Code is *tokenized* by set of of lexical rules (Lexical Analysis)

Token sequence is *transformed/normalized* by a set of rules

Token Based Clone Detection

Code is *tokenized* by set of of lexical rules (Lexical Analysis)

→

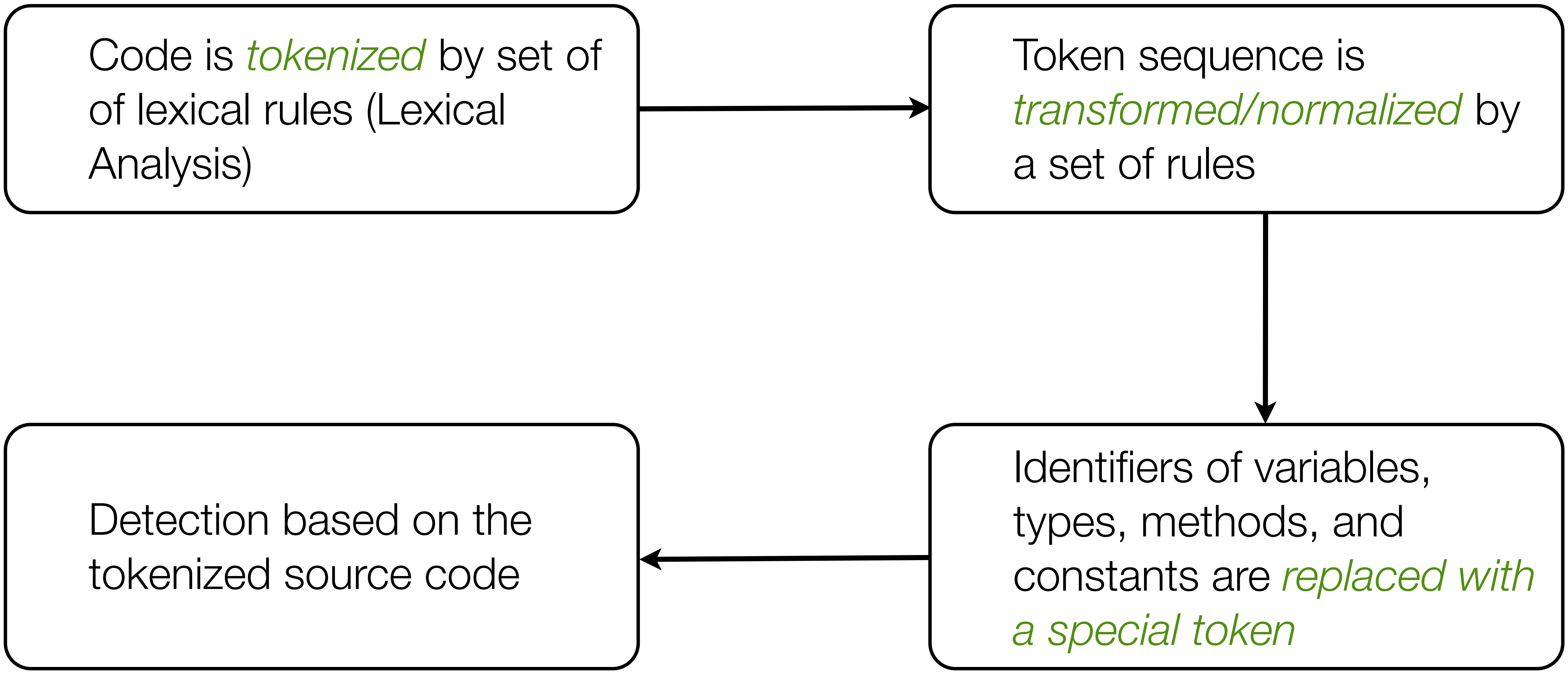
Token sequence is *transformed/normalized* by a set of rules

↓

Identifiers of variables, types, methods, and constants are *replaced with a special token*

Code segments with different variable names become clone pairs

Token Based Clone Detection



Code segments with different variable names become clone pairs

Token Based Clone Detection

Token Based Clone Detection

```
Void print_lines(const set<string>& s) {
    int c = 0;
    set<string>::const_iterator i
        = s.begin();
    for (; i != s.end(); ++i) {
        cout << c << ", "
            << *i << endl;
        ++c;
    }
}
```

Token Based Clone Detection

```
Void print_lines(const set<string>& s) {
    int c = 0;
    set<string>::const_iterator i
        = s.begin();
    for (; i != s.end(); ++i) {
        cout << c << ", "
            << *i << endl;
        ++c;
    }
}
```

```
void print_lines ( const set & s ) {
int c = 0 ;
Const_iterator I
= s . begin ( ) ;
for ( ; i != s . end ( ) ; ++ i ) {
cout << c << ", "
<< * I << endl ;
++ c ;
}
}
```

Token Based Clone Detection

```
Void print_lines(const set<string>& s) {
    int c = 0;
    set<string>::const_iterator i
        = s.begin();
    for (; i != s.end(); ++i) {
        cout << c << ", "
            << *i << endl;
        ++c;
    }
}
```

```
void print_lines ( const set & s ) {
int c = 0 ;
Const_iterator I
= s . begin ( ) ;
for ( ; i != s . end ( ) ; ++ i ) {
cout << c << ", "
<< * I << endl ;
++ c ;
}
}
```

```
$p $p ($p $p & $p ) {
$p $p = $p ;
$p $p
= $p . $p ( ) ;
for ( ; $p != $p . $p ( ) ; ++ $p ) {
$p << $p << $p
<< * $p << $p ;
++ $p ;
}
}
```

Metrics Based Clone Detection Mayrand'96

- Basic idea: For a given code segment a *metrics profile* is calculated
- Basic assumption: Similar code segments have similar metrics profiles
- Granularity: Method/Function level
- Approach makes sense: Most segments are copied & pasted (and slightly modified)
- But they keep their basic properties

Metrics Based Clone Detection Mayrand'96

- 4 Points of comparison:
  - Name
  - Layout
  - Expressions
  - Control flow
- Each point is a set of numerical metrics describing certain aspects of a method/function

1. Point of comparison: Name

“If two functions have the same name they are likely clones.”

Relative number of common characters

2. Point of comparison: Layout

Layout is defined as: "*the visual organization of the source code*"

Count the number of comments, number of non-blank lines, average length of variable names, ...

3. Point of comparison: Expressions

"***the number of expressions in a function, their nature and their complexity are considered***"

Count calls to other methods, number of declaration statements, number of executable statements, conditional complexity

4. Point of comparison: Control Flow

"***the control flow characteristics of a method***"

Number of unique paths, number of loops, nesting level, number of exits, number of conditional decisions, ...

Scale	Nam	Lay	Exp	Con
1-ExactCopy	=	=	=	=
2-DistinctName	!=	=	=	=
3-SimilarLayout	X	~	=	=
4-DistinctLayout	X	!=	=	=
5-SimilarExpression	X	X	~	=
6-DistinctExpression	X	X	!=	=
7-SimilarControlFlow	X	X	X	~
8-DistinctControlFlow	X	X	X	!=

Symbol	Description
=	Equal values for all metrics in a point of comparison between two functions
~	At least one metric not equal but within the delta in a point of comparison between two functions
!=	At least one metric not equal and outside the delta in a point of comparison between two functions
X	The point of comparison is not considered in the scale evaluation.

Predefined Code Clone Classes

Clone Visualization

Software systems in practice are large thousands of files

Cloned code exists in different files, different methods of the same files

How can we presents the results of clone detection?

Quick and efficient overview where similarities in source code occur

Dot Plot Visualization

Adopt idea for *DNA Analysis*: Compare protein and DNA sequences

Dot plot is an established technique to *compare sequences*

(not to be confused with dot plots from statistics)

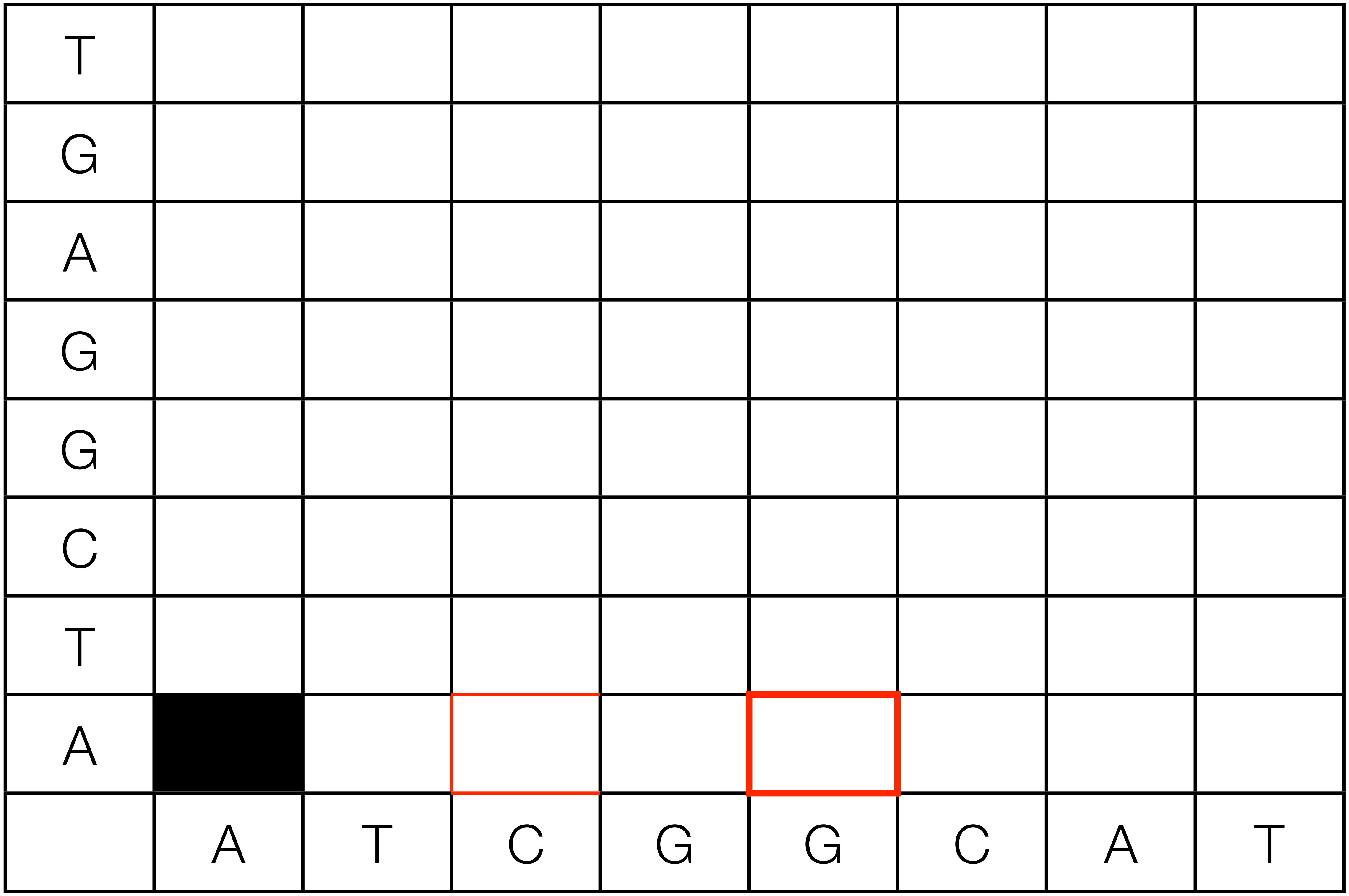
T								
G								
A								
G								
G								
C								
T								
A								
	A	T	C	G	G	C	A	T

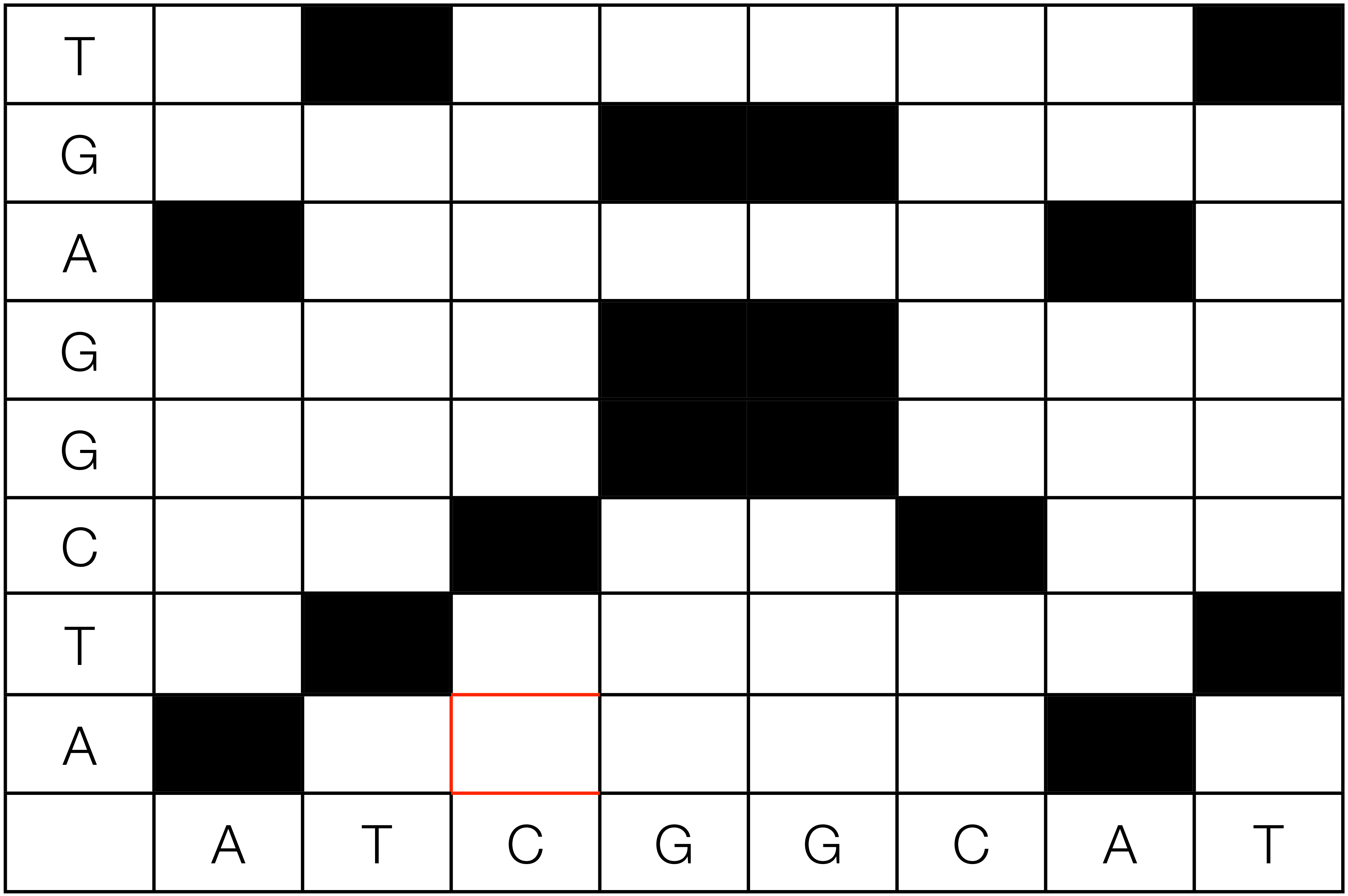
T								
G								
A								
G								
G								
C								
T								
A	■							
	A	T	C	G	G	C	A	T

T								
G								
A								
G								
G								
C								
T								
A								
	A	T	C	G	G	C	A	T

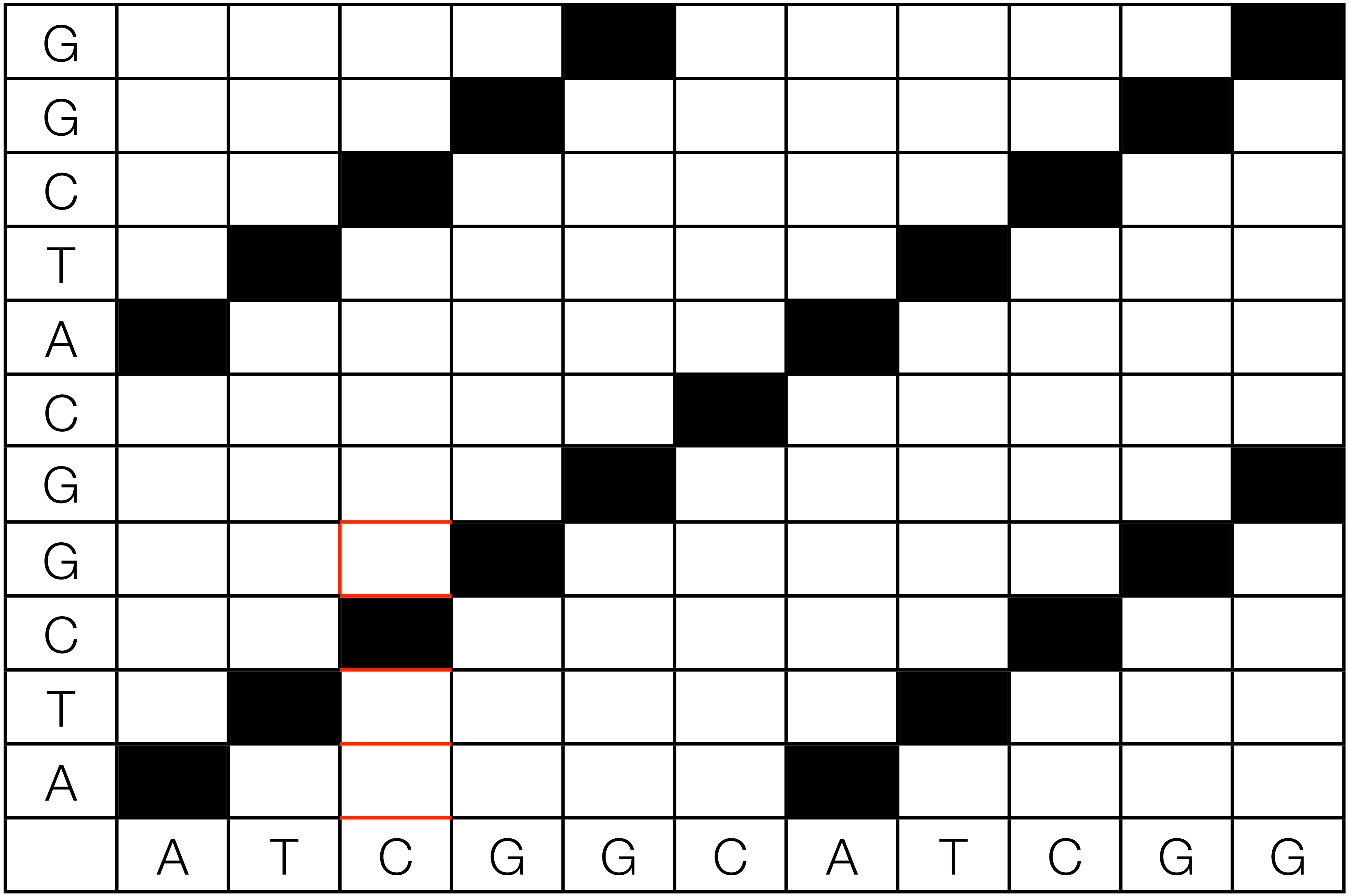
T								
G								
A								
G								
G								
C								
T								
A								
	A	T	C	G	G	C	A	T

T								
G								
A								
G								
G								
C								
T								
A								
	A	T	C	G	G	C	A	T


T
G
A
G
G
C
T
A
A
T
C
G
G
C
A
T



(No sequence length restrictions)



(Clone sequence length > 2)

Dot Plot Visualization

- Code (according to granularity) is put on vertical / horizontal axis
- A match between two elements is a dot in the matrix
- Easy visual identification of insertion, deletions, repeats, variations

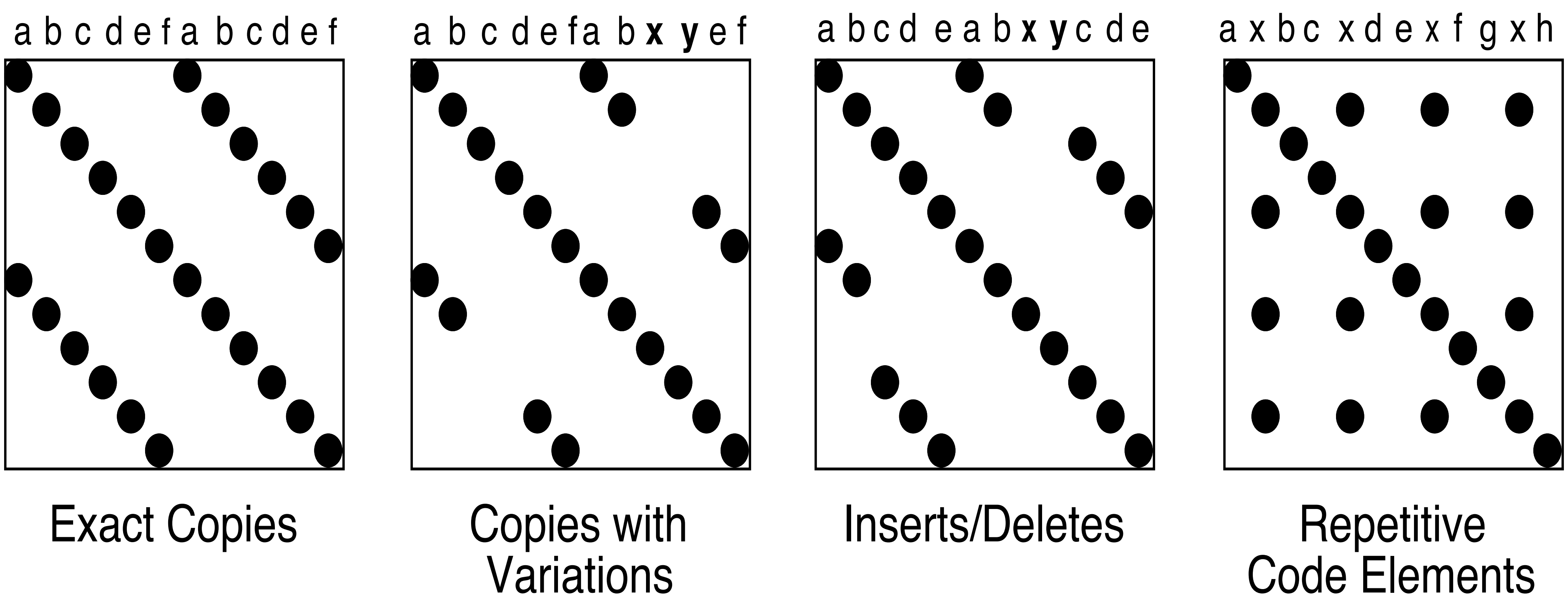
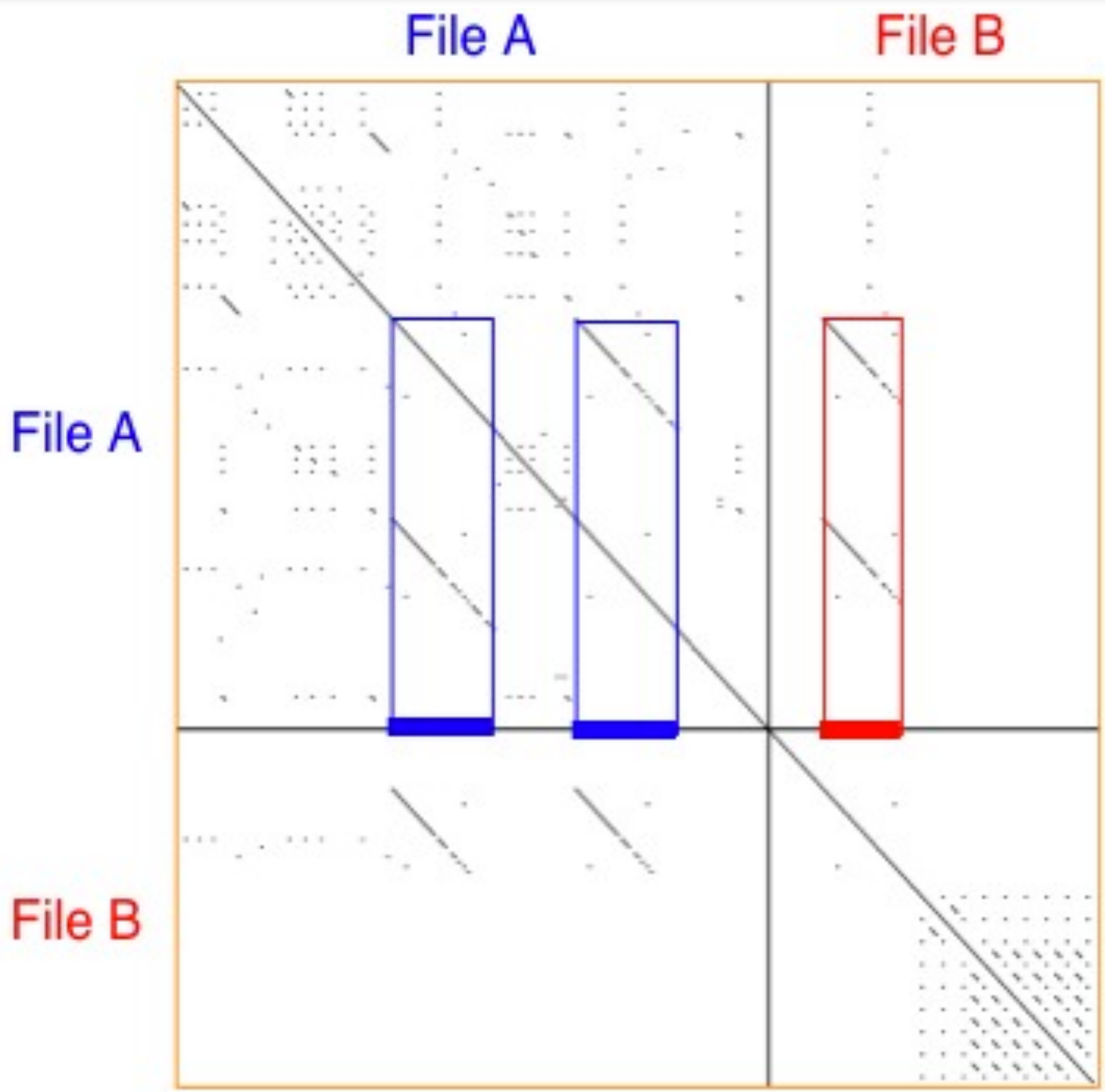


Image by Uni of Berne [Rieger & Ducasse 1999]

Example: Copied Code Sequences

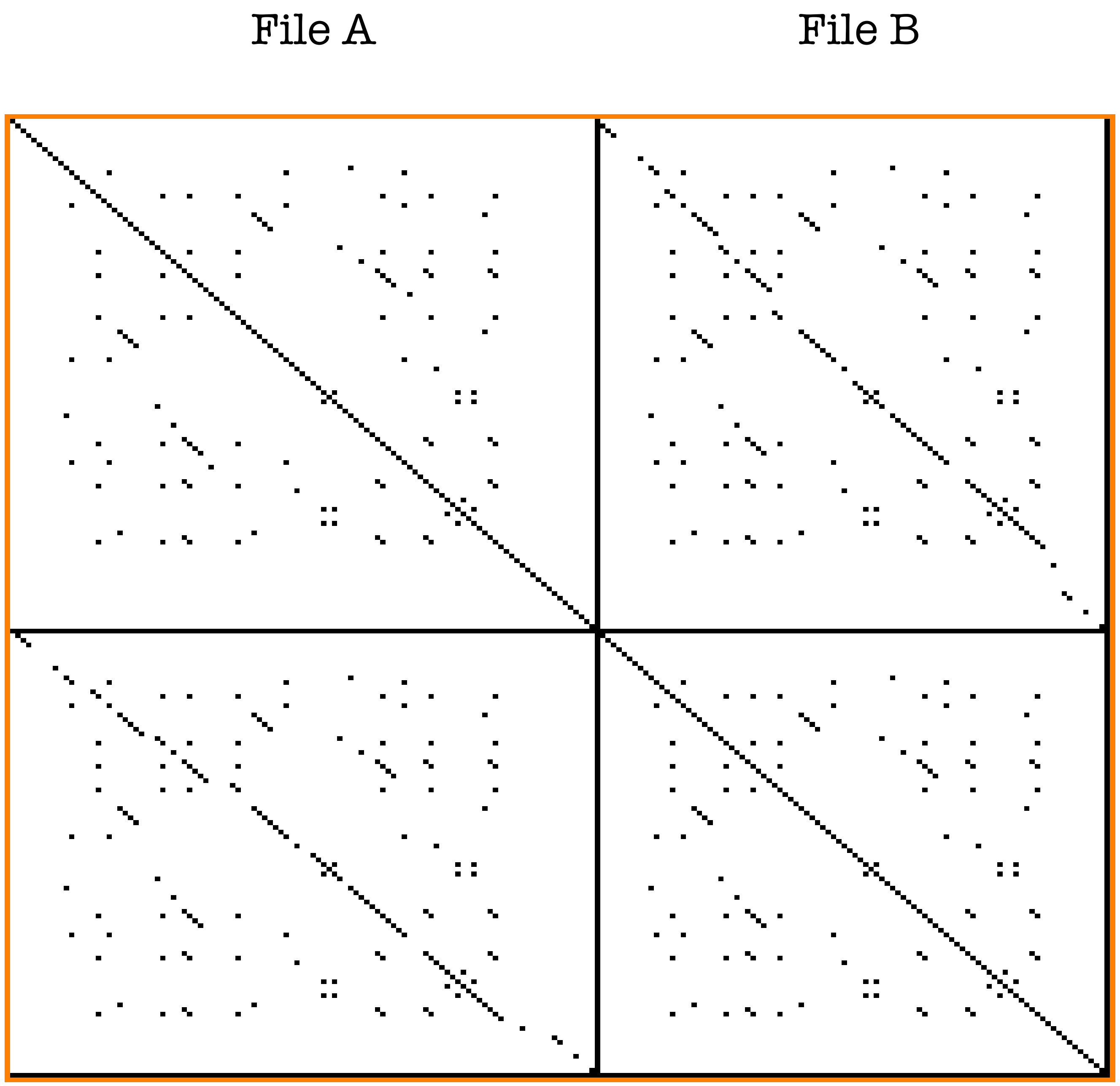
- File A contains two copies of a code segment
- File B also contains two copies of that segment
- Extract Method to refactor clone?
- Examples are made using *Duploc* from an industrial case study (1 Mio. LOC C++ System)
- Duploc @ Uni Berne [Rieger & Ducasse 1999]



Example: Cloned Class

One Class is an edited copy of another class

Subclassing to refactor clones?



Dot Plots

+ Good overall impression

+ Easy to spot patterns

- Pairwise comparison

- Scalability?

Conclusion

- Code Clones exist and *can* be problematic during maintenance (*inconclusive results from research*!)
- Solution: *Periodic clone assessment* of a software, e.g., major releases,
- Detection of clones is nontrivial
- Efficient visualization of code clones is needed for real world system
- There is *tool support*